Compositional Verification of Evolving SPL

Jean-Vivien Millo, S. Ramesh, Shankara Narayanan Krishna, Ganesh Khandu Narwane

Compositional Verification of Evolving SPL

Jean-Vivien Millo, S Ramesh, Shankara Narayanan Krishna, Ganesh Khandu Narwane

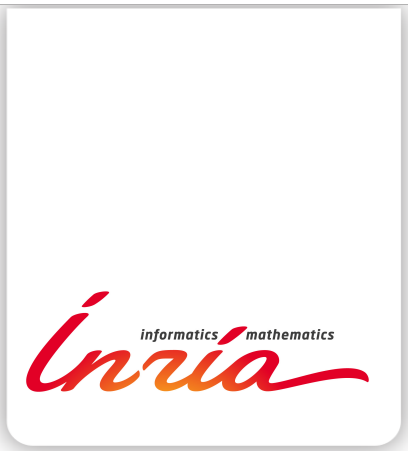
RESEARCH REPORT

N° 8125

October 2012

Project-Teams AOSTE

ISSN 0249-6399 ISRN INRIA/RR--8125--FR+ENG



Compositional Verification of Evolving SPL

Jean-Vivien Millo, S Ramesh*, Shankara Narayanan Krishna†,
Ganesh Khandu Narwane‡

Project-Teams AOSTE

Research Report n° 8125 — October 2012 — 31 pages

Abstract: This paper presents a novel approach to the design verification of Software Product Lines(SPL). The proposed approach assumes that the requirements and designs are modeled as finite state machines with variability information. The variability information at the requirement and design levels are expressed differently and at different levels of abstraction. Also the proposed approach supports verification of SPL in which new features and variability may be added incrementally. Given the design and requirements of an SPL, the proposed design verification method ensures that every product at the design level behaviorally conforms to a product at the requirement level. The conformance procedure is compositional in the sense that the verification of an entire SPL consisting of multiple features is reduced to the verification of the individual features. The method has been implemented and demonstrated in a prototype tool SPLEnD (SPL Engine for Design Verification) on a couple of fairly large case studies.

Key-words: Design verification, Software Product Line, SPIN, QSAT

* Global General Motors R&D, TCI Bangalore, India
† Department of CSE, IIT Bombay, Mumbai, India
‡ Homi Bhabha National Institute, Mumbai, India

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Compositional Verification of Evolving Software Product Lines

Résumé : Ce papier présente une approche nouvelle de vérification pour les lignes de produits logiciels (LPL). L'approche proposée considère que la spécification et la conception de LPL peuvent être abstraites comme des automates à états finis comprenant des informations sur la variabilité. Ces informations sont exprimées différemment aux niveaux spécification et conceptions. Sous ces hypothèses, l'approche proposée supporte la vérification de LPLs dans lesquelles des fonctionnalités peuvent être ajoutées incrémentalement.

A partir de la spécification et de la conception d'une LPL, la méthode de vérification proposée assure que chaque produit au niveau conception se conforme, comportementalement parlant, à un produit au niveau spécification.

La procédure de conformité est compositionnelle car la vérification de la LPL en entier se réduit à la vérification des fonctionnalités qui la compose individuellement. La méthode a été implantée dans un outil appelé "SPLEnD" et essayée sur deux cas d'étude relativement larges.

Mots-clés : Vérification, Ligne de produits logiciels, SPIN, QSAT

1 Introduction

Large industrial software systems are often developed as *Software Product Line* (SPL) with a common core set of features which are developed once and reused across all the products. The products in an SPL differ on a small set of features which are specified using *variation points*. The focus of this paper is on modeling and analysis of SPLs which have drawn the attention of researchers recently [1, 3, 4].

Many approaches have been proposed to describe SPLs, the most prominent one being *feature diagrams*. All these proposals seem to assume a global view of SPL as they start with a complete list of features and the variation points using a single vocabulary. All the subsequent SPL assets, like requirement documents, design models, source codes, test cases, documentations, share the same definition and vocabulary [6, 18]. The assumption of a single homogeneous and global view of variability description is inapplicable in many practical settings, where there is no top level complete description of features and variabilities. They often evolve during the long lifetime of an SPL as new features and variabilities are added during the evolution. Further, SPL developers tend to use different representations and vocabulary of variability at different stages of development: at the requirement level, a more abstract and intuitive description of variation points are used, while at the design level, the efficiency of implementation of variation points is of primary concern. For example, consider the case of an automotive SPL, where one variation point is the region of sale (eg. Asia Pacific, Europe, North America etc). At the requirement level, this variation point is expressed directly as an enumeration variable assuming one value for every region. Whereas, at the design level, the variation point is expressed using two or three boolean variables; by setting the values of the boolean variable appropriately, the behavior specific to a region is selected at the time of deployment.

We present a design verification approach that is more suited to the above kind of evolving SPLs in which different representation of variabilities would be used at the requirement and design level. One natural and unique problem that arises in this context is to relate formally the variation points expressed at different levels of abstractions. Another challenge is the analysis complexity: the number of products is exponential in the number of variation points and hence product centric analyses are not scalable. We propose a compositional approach in which every feature of the SPL is first analyzed independently; the per-feature analysis results are then combined to get the analysis result for the whole SPL.

For capturing variability in the behavior of an SPL, we have extended the standard finite state machine model, which we call *Finite State Machines with Variability*, in short, *FSMv*. The behavior and variability of a feature at the requirement and design level can be modeled using FSMv. We define a conformance relation between FSMvs to relate the requirement and design models. This relation is based upon the standard language containment of state machines.

One unique feature of FSMv is that it provides a compositional operator

for composing the feature state machines to obtain a model for an SPL. This operator thus enables incremental addition of features and variabilities. The proposed verification approach exploits the compositional structure of the SPL models to contain the analysis complexity.

Figure 1 summarizes the proposed approach. It shows an SPL composed of features f_1 to f_n. Each feature has an FSMv model of its requirements (called FSMr) and an FSMv model derived from its design (called FSMd). The proposed analysis method checks whether the FSMd of every feature conforms to its FSMr (1^{st} check). The output of this first step is a conformance relation between each pair of FSMr and FSMd. The obtained conformance relations are then used to check whether the actual behavior of the entire SPL conforms to the expected one (2^{nd} check). We reduce this check to checking the satisfiability of a Quantified Boolean Formula (QBF). There is no need to build the entire behavioral model of the SPL in the second step.

We have built a prototype tool SPLEnD based upon this approach. This tool performs the first check using SPIN [13] while the the well-known QBF SAT solver CirQit [10] is used for the second step. We have experimented with the tool using modest industrial size examples with very encouraging results.

1.1 Related works

FSMv and the proposed design verification approach were developed independently but has some apparent similarities with the FTS$^+$ model [3], which also extends finite state machines to include certain product variability information. However, there is a motivational difference between the two formalisms. The aim of FTS$^+$ is to model the entire SPL and hence there is a single global machine with a single global vocabulary for expressing variabilities; the variability information represents the presence/absence of features in the SPL. In contrast, our approach is based upon a differnt view of SPL: a feature with variability is an increment in functionality and an SPL is a collection of features. We use a single FSMv to model a feature and a whole SPL is modeled as a parallel composition of FSMv machines.

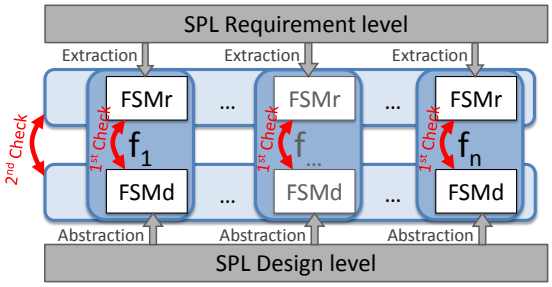


Figure 1: The proposed verification framework.

The difference in viewpoint has another consequence: FTS$^+$ models, since they model the entire SPL, tend to be large and hence has high analysis complexity. Efficient abstraction techniques are hence used for solving this problem [4]. Whereas, each FSMv models a fraction of functionality and hence can be analysed easily. Further, the entire SPL can be modeled as composition of FSMvs and can be efficiently analysed using composition techniques.

Many other behavioral models have also been proposed [15, 20, 7, 11] which are usually coupled with a variability model such as OVM [18], the Czarnecki feature model [6], or VPM [9] to attain a fair level of variability expressibility. Unlike all these approaches, FTS$^+$ [3] and FSMv capture the variability in an explicit way which we find more intuitive.

The Variation Point Model (VPM) of Hassan Gomaa [9] distinguishes between variability at the requirement and design levels but no design verification approach has been presented. Kathrin Berg *et al.*[2] propose a model for variability handling throughout the life cycle of the SPL. Andreas Metzeger *et al.*[19] and M Riebisch *et al.*[21] provide a similar approach but they do not consider the behavioral aspect. In the proposed approach, we extract the relation between requirement and design level variability from a behavioral analysis.

Kathi Fisler *et al.* [14] have developed an analysis based on three-valued model checking of automata defined using step-wise refinement. Later on, Jing Liu *et al.* [17] have revisited Fisler's approach to provide a much more efficient method. Recently, Maxime Cordy *et al.* have extended Fisler's approach to LTL formula [5]. Kim Lauenroth *et al.* [16] as well as Andreas Classen *et al.* [3, 4], and Gruler *et al.* [12] have developed model checking methods for SPL behavior. These methods are based on the verification of LTL/CTL/modal μ calculus formula.

All these verification methods assume a global view of variability and hence the representation of variability information is identical in both specification and the design. In contrast, in our work the specification and design involve variability information at different levels of abstraction and hence one needs mapping information between the two levels. Furthermore, our formalism allows incremental addition of functionality and variability and enables compositional verification.

2 Design Verification of a Single Feature

An SPL, in general consists of multiple features, each feature having different functionality and variability. A typical body control software of an automotive system is an SPL that has several features such as door lock, lighting, seat control etc. Each of these features has a distinct function and variability. For example, the locking behaviour of a door lock function has a variation point called *transmission type*. If the transmission type is manual then the door is locked after the speed of the vehicle exceeds a certain threshold value; for automatic transmission, the door is locked when the gear position is shifted out of park. In this section we will focus on modeling and relating the design of a

single feature to its requirement.

2.1 FSMv and language refinement

Finite State Machines with Variability (FSMv) is an extension of finite state machines, to represent all possible behaviours of a feature. Let Var be a finite set of variables, each taking a value ranging over a finite set of values. Let $x \in Var$, and let $Dom(x)$ be the finite set of values that x can take. Let $S \subseteq Dom(x)$. The set of atomic formulae we consider are $x = a$, $x \neq a$, $x \in S$, $x \notin S$ for $a \in Dom(x)$ and $x = y$, $x \neq y$ for $x, y \in Var$. Let A_{Var} denote the set of atomic formulae over Var. Let α represent a typical element of A_{Var}. Define $\Delta ::= \alpha \mid \neg\Delta \mid \Delta \wedge \Delta \mid \Delta \vee \Delta \mid \Delta \Rightarrow \Delta$ to be the set of all well formed predicates over Var.

Definition 1 (FSMv)**.** *An FSMv is a tuple $\mathcal{A} = \langle Q, q_0, \Sigma, Var, E, \rho \rangle$ where: (1) Q is a finite set of states; q_0 is the initial state; (2) Σ is a finite set of events; (3) Var is a finite set of variables; (4) $E \subseteq Q \times \Delta \times \Sigma \times Q$ gives the set of transitions. A transition $t = (s, g, a, s')$ represents a transition from state s to state s' on event a; the predicate g is called a guard of the transition t; g is consistent and defines the variability domain of the transition; (5) $\rho \in \Delta$ is a consistent predicate called the global predicate.*

The variables in Var determines the variability allowed in the feature with each possible valuation of the variables corresponding to a variant. The allowed values of the variables are constrainted by the global predicate ρ. For example, if ρ is $((x = 1) \vee (x = 2)) \wedge (x = y - 1)$, then the allowed variants are those for which the values for the pairs (x, y) are $(1, 2), (2, 3)$. The predicate in a transition determines the variants to which the transition is applicable. While drawing a transition $t = (s, g, a, s')$, the edge connecting s to s' is decorated with $g : a$. When g is true, we simply write a on the edge.

Definition 2 (Configuration)**.** *A configuration, denoted by π, is an assignment of values to the variables in Var. The set of all configurations is denoted by Π_{Var}, or Π, when Var is clear from the context. Define $\Pi(\rho) = \{\pi \mid \pi \models \rho\}$ to be the set of all those configurations that satisfy ρ. The elements of $\Pi(\rho)$ are called valid configurations. Given a valid configuration π and a transition $t = (s, g, a, s')$, we say that t is enabled by π if $\pi \models g$.*

As a concrete example of an FSMv, consider the feature *Door lock* in automotive SPL which controls the locking of the doors when the vehicle starts. The expected behavior of this feature is modeled using the FSMv Req_{dl} described pictorially in Figure 2. In the initial state, this feature becomes active when all the doors are closed. The doors are locked when either the speed of the vehicle exceeds a predefined value or the gear is shifted out of park. An unlock event reactivates the feature. There are four configurations for this feature all of which are described using the three variables: DL_Enable, $Transmission_{dl}$ and DL_User_Pref. The top box denotes the values that these variables

can assume, and the bottom box gives the global predicate (ρ) associated with the machine. ρ ensures that in every valid configuration, the variable $Transmission_{dl}$ having the value $Manual$ implies that DL_User_Pref takes the value *Speed*. This captures the fact that in manual transmission, there is no park position on the gearbox. To avoid clutter, we have replaced guards of the form $x = i$ with i in the figure. The transition labeled with $Disable : *$ means that when DL_Enable assumes the value $Disable$, it stalls on any event.

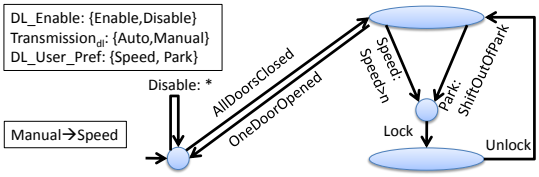


Figure 2: The FSMv of the feature *Door lock*.

2.1.1 Requirement against Design

In the requirement of a product line, the variability is usually discussed in terms of variation points, which are at a high level of abstraction and focused on clarity and expressibility. The restriction of the possible configurations is expressed as general constraints on these variation points, e.g., the global predicate $Manual \implies Speed$ in the *Door lock* example. In contrast, in a design, the variability description is constrained by efficiency, implementability, ease of reconfiguration and deployment considerations. For instance, in the automotive applications, one often finds *calibration parameters* ranging over a set of boolean values. Further, the constraint on the calibration parameters (ρ) takes the special form of the list of the possible configurations of the calibration parameters in order to easily configure the design.

FSMv can capture both the design as well as the requirements of a feature. We distinguish the requirement and design models by denoting them FSMr and FSMd respectively. Figure 2 presents the FSMr, Req_{dl}, of the feature *Door lock*. The FSMd, Des_{dl}, of the feature *Door lock* is presented in Figure 3. The structure of Des_{dl} is similar to Req_{dl} except that the top elliptical shaped state in Figure 2 is split into two states (the top and the bottom elliptical shaped states) in Figure 3. The top state is for auto-transmission whereas the bottom one is for manual transmission as can be seen from the configuration label of the two transitions going from the initial state. Two variables $Cp1$ and $Cp2$ encode the possible configurations in the FSMd. The box in Figure 3 depicts the set of possible values of these. $Cp1 = Auto$ corresponds to the configuration in which the transmission is $Auto$ whereas $Cp1 = Moff$ corresponds to either the manual transmission or the case when $Cp1$ is disabled; similarly, $Cp2 = Speed$ means that the user preference is set on *Speed*, while $Cp2 = Poff$ means either $Park$ or the case when $Cp2$ is disabled.

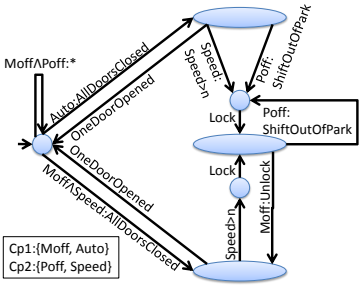


Figure 3: Des_{dl}: the FSMd abstracted from the design of the feature *Door lock*.

2.2 Variants of FSMv and Conformance

Having described the design and requirement behaviour of a feature f using FSMd and FSMr respectively, we now define the notions of variants and conformance. A variant of an FSMv corresponds to one of the several possible behaviours of the feature (at the design, requirement level respectively). Given a feature f, and a (FSMd, FSMr) pair corresponding to f, we say that the design of f conforms to the requirements of f provided every variant of the FSMd has a corresponding FSMr variant.

Definition 3 (Variant of an FSMv)**.** *Let $\mathcal{A} = \langle Q, q_0, \Sigma, Var, E, \rho\rangle$ be an FSMv and $\pi \in \Pi(\rho)$ be a valid configuration of $\mathcal{A}$. A variant of $\mathcal{A}$ is an FSM obtained by retaining only transitions $t = (s, g, a, s')$, and states s, s' such that $g \models \pi$. Once the relevant states and transitions are identified, we remove the guards g from all the transitions; ρ is also removed. The resultant FSM is denoted $\mathcal{A} \downarrow \pi$.*

In the example of FSMr for the feature *Door lock*, the variant $Req_{dl} \downarrow \langle Enable, Auto, Park\rangle$ does not contain the transitions with the event $Speed > n$ and $*$. We compare the FSMd and FSMr of a feature f using their variants. Given an FSMv $\mathcal{A}$, we associate with each configuration π of $\mathcal{A}$ the language of the FSM $\mathcal{A} \downarrow \pi$, denoted by $L(\mathcal{A} \downarrow \pi)$. We say that an FSMd $\mathcal{A}_d$ conforms to an FSMr $\mathcal{A}_r$ if and only if the behaviour of every variant of $\mathcal{A}_d$ is contained in the behaviour of some variant of $\mathcal{A}_r$.

Definition 4 (The conformance mapping Φ)**.** *Let $\mathcal{A}_r$ and $\mathcal{A}_d$ be a pair of FSMr and FSMd respectively with global predicates ρ^d and ρ^r. Let Π_d, Π_r be the set of all design, requirement configurations. Then $\mathcal{A}_d$ conforms to $\mathcal{A}_r$ denoted $\mathcal{A}_d \leq_\Phi \mathcal{A}_r$ if there exists a mapping $\Phi : \Pi_d(\rho^d) \to 2^{\Pi_r(\rho^r)}$ such that $\forall \pi_d \in \Pi_d(\rho^d), \exists \pi_r \in \Pi_r(\rho^r)$ satisfying $L(\mathcal{A}_d \downarrow \pi_d) \subseteq L(\mathcal{A}_r \downarrow \pi_r)$. Φ is called the conformance mapping.*

In the feature *Door lock*, $\Phi(\langle Moff, Speed\rangle)$ contains $\langle Enable, Manual, Speed\rangle$ since $L(Des_{dl} \downarrow \langle Moff, Speed\rangle) \subseteq L(Req_{dl} \downarrow \langle Enable, Manual, Speed\rangle)$.

2.3 Checking the conformance

Let f be a feature with FSMr Req_f and FSMd Des_f. Then the conformance checking problem is to compute a mapping Φ such that $Des_f \leq_\Phi Req_f$.

The conformance mapping is computed by comparing every projection of Des_f with every projection of Req_f. Algorithm 1, given below, presents a possible implementation using the standard automata containment algorithm[22], as implemented in the SPIN model checker [13]. To use SPIN, one should describe the system along with the checked property in the Promela language [13]. Out of this description, SPIN generates the *pan.c* file which is the verifier for the system. After compilation, the *pan(.exe)* executable performs the verification. The details of the conformance checking using Algorithm 1 and its correctness can be found in Appendix A. Lemma 5 proves the correctness of Algorithm 1.

Algorithm 1 implements the conformance checking using SPIN.

Input : Des_f, Req_f.
Output : The mapping Φ when $Des_f \leq_\Phi Req_f$
1. Generate a Promela file which contains Req_f, Des_f, the environment, the never claim, and the initialization sequence.
2. Launch the full verification algorithm of spin
3. Build the mapping Φ from the output of spin.
4. Conclude whether the design conforms to the requirement
if $\forall \pi_d \in \Pi(\rho_d)$, $\Phi(\pi_d) \neq \emptyset$ **then**
  return *true* along with (Φ)
else
  return $false$ along with (π_d) {where π_d has no correspondence through Φ}
end if

Lemma 5. *Given FSMd Des_f and FSMr Req_f for a feature f, let (π_d, π_r) be a pair of design and requirement configurations. Then, $L(Des_f \downarrow \pi_d) \not\subseteq L(Req_f \downarrow \pi_r)$ if and only if $\neg error(Des_f) \wedge error(Req_f)$.*

Proof. The proof can be found in Appendix A.1. □ □

3 Design Verification of SPL

In the previous section, we looked at individual features in an SPL and provided a method for comparing the design and requirements of a feature, both containing variabilities. In this section, we extend this method to verifying a whole SPL design against its requirements. An SPL is essentially a composition of multiple features satisfying certain constraints. We define a parallel composition operator over FSMv to model an SPL. The features in an SPL can interact and we follow one of the standard methods of allowing the composed FSMv models to share some common events, which correspond to two-party handshake communication events. A distinguishing aspect of the proposed parallel operator is

that it takes into account the constraints across the composed machines. The constraints could be of various types, e.g. dependency and exclusion relations, and are modeled as predicates over variables of the composed features.

Definition 6 (Parallel composition of FSMv)**.**
Let $\mathcal{A}_x = \langle Q_x, q_0^x, \Sigma_x, Var_x, E_x, \rho_x \rangle$, $x \in \{1, 2\}$ *be two FSMv's with* $Var_1 \cap Var_2 = \emptyset$. *Let* $H = \Sigma_1 \cap \Sigma_2$ *be the set of handshaking events. Let* ρ_{12} *be a predicate over* $Var_1 \cup Var_2$, *such that* $\rho_{12} \wedge \rho_1 \wedge \rho_2$ *is consistent.* ρ_{12} *is the composition predicate capturing the possible constraints between the variabilities of the two composed features. Let* $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$.

The parallel composition of $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *denoted by* $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ *is a tuple* $\langle Q_1 \times Q_2, (q_0^1, q_0^2), \Sigma_1 \cup \Sigma_2, Var_1 \cup Var_2, E, \rho \rangle$ *with transitions defined as follows: Consider a state* $(s_1, s_2) \in Q_1 \times Q_2$, *and transitions* $(s_1, g_1, a_1, s_1') \in E_1$ *and* $(s_2, g_2, a_2, s_2') \in E_2$.
(1) If $a_1 = a_2 = a \in H$, *define* $((s_1, s_2), g_1 \wedge g_2, a, (s_1', s_2')) \in E$, *provided* $g_1 \wedge g_2$ *is consistent and* $g_1 \wedge g_2 \models \rho$.
(2) If $a_1 \in \Sigma_1 \backslash H$, *define* $((s_1, s_2), g_1, a_1, (s_1', s_2)) \in E$, $g_1 \models \rho$.
(3) If $a_2 \in \Sigma_2 \backslash H$, *define* $((s_1, s_2), g_2, a_2, (s_1, s_2')) \in E$, $g_2 \models \rho$.

For illustration, consider the feature *Door unlock* which automates the unlocking of the doors in a vehicle. Figure 4-a gives the FSMr of the feature extracted from the requirements. From the initial state, the feature becomes active when the event *Lock* happens. As soon as either the key is removed from ignition or the gear is shifted to park position, the doors get unlocked and the feature *Door unlock* becomes inactive. Figure 4-b presents the FSMd of the feature *Door unlock*. It is quite similar to the requirement except that the active state is split in two: the feature reacts to the *ignition Off* event in one state, and to the *Shift Into Park* event in another state.

Let us consider the composition of the two FSMrs of the features *Door lock* and *Door unlock*. The handshake events between the two features are *Lock* and *Unlock*. In the composition, we introduce the following composition predicate: $(DU\_Enable = Enable \Leftrightarrow DL\_Enable = Enable) \wedge Transmission_{dl} = Transmission_{du}$, which brings out the natural constraints that *Door lock* feature is enabled if and only if *Door unlock* is also enabled and the transmission status has to be the same.

The valid configurations after composition are restricted by the composition predicate. We provide a few definitions to define composite valid configurations.

Definition 7 (Composing Configurations)**.** *Let* $\mathcal{A}_i = (Q_i, q_0^i, \Sigma_i, Var_i, E_i, \rho_i)$ *be two FSMv's, and let* $\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2$ *be as given by definition 6. Let* $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$ *be the global predicate of* $\mathcal{A}$. *Consider two valid configurations* $\pi_1 \in \Pi(\rho_1)$ *and* $\pi_2 \in \Pi(\rho_2)$ *of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$. *The compostion of* π_1, π_2, *denoted* π_{12} *is a configuration over* $Var_1 \cup Var_2$ *such that* π_{12} *agrees with* π_1 *over* Var_1, *and agrees with* π_2 *over* Var_2, *and* $\pi_{12} \models \rho$. π_{12} *is a valid configuration of* $\mathcal{A}$ *and we denote it by* $\pi_{12} = \pi_1 + \pi_2$.

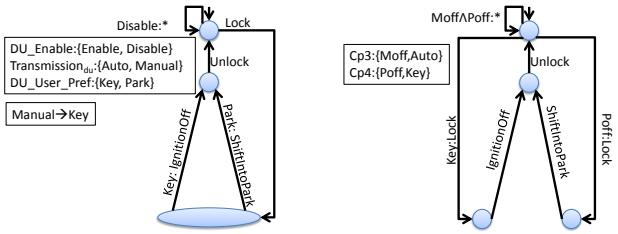


Figure 4: a) Req_{du}: the *Door unlock* FSMr and b) Des_{du}: the corresponding FSMd.

Lemma 8. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two FSMv's. For each valid configuration π of $\mathcal{A}_1 \parallel \mathcal{A}_2$, there are valid configurations π_1 of $\mathcal{A}_1$ and π_2 of $\mathcal{A}_2$ such that $\pi = \pi_1 + \pi_2$.*

Proof. In Appendix B. □ □

In the example of feature *Door Lock*, the configuration $\langle Enable, Auto, Speed\rangle$ from Req_{dl} can be composed with $\langle Enable, Auto, Key\rangle$ from Req_{du} because the transmission is *Auto* in both (which is specified in the composition predicate). $\langle Enable, Auto, Speed, Enable, Auto, Key\rangle$ is a configuration of the parallel composition of Req_{dl} with Req_{du}.

The parallel composition of FSMv's is such that each variant of the composition of two FSMv's is equal to the composition of variants of the individual FSMv's.

Lemma 9 (Variants of a composed FSMv)**.** *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two FSMv machines. Let π be a valid configuration of $\mathcal{A}_1 \parallel \mathcal{A}_2$. Then $L([\mathcal{A}_1 \parallel \mathcal{A}_2] \downarrow \pi) = L(\mathcal{A}_1 \downarrow \pi) \parallel L(\mathcal{A}_2 \downarrow \pi)$.* [1]

Proof. In Appendix C. □ □

3.0.1 Refinement and Parallel Composition

The definition of parallel composition naturally lends itself to a notion of addition of conformance mappings between design and requirement pairs. Consider FSMr's R_1, R_2 corresponding to two features f_1, f_2. Let D_1, D_2 be the corresponding FSMd's. Let ρ_1^r, ρ_2^r be the global predicates of R_1, R_2, and let ρ_1^d, ρ_2^d be the global predicates of D_1, D_2 respectively. Assume that $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$. Let $\rho^r = \rho_{12}^r \wedge \rho_1^r \wedge \rho_2^r$ be the global predicate of $R_1 \parallel R_2$; likewise, let $\rho^d = \rho_{12}^d \wedge \rho_1^d \wedge \rho_2^d$ be the global predicate of $D_1 \parallel D_2$. We now want to ask if $D_1 \parallel D_2$ conforms to $R_1 \parallel R_2$. This amounts to computing a conformance

[1] The right hand side $\parallel$ refers to the standard communicating finite state machine composition.

mapping between $D_1 \parallel D_2$ and $R_1 \parallel R_2$ given Φ_1, Φ_2. Consider any valid configuration π^d of $D_1 \parallel D_2$. By Lemma 8, we can write π^d as $\pi_1^d + \pi_2^d$, where π_1^d, π_2^d are valid configurations of D_1, D_2 respectively. Since $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$, there exists valid configurations $\pi_1^r \in \Phi_1(\pi_1^d)$ and $\pi_2^r \in \Phi_2(\pi_2^d)$ in R_1, R_2 respectively. Given this, the addition of Φ_1, Φ_2 is defined as follows:

Definition 10 (Addition of conformance mappings)**.** *The addition of conformance mappings Φ_1, Φ_2 is defined to be a mapping $\Phi = \Phi_1 + \Phi_2$ as follows. For every valid configuration $\pi^d = \pi_1^d + \pi_2^d$ of $D_1 \parallel D_2$,*

$$\Phi(\pi^d) = \{\pi^r \mid \pi^r \textit{ is a valid configuration of } R_1 \parallel R_2, \pi^r = \pi_1^r + \pi_2^r \textit{ for valid configurations } \pi_1^r \in \Phi_1(\pi_1^d), \pi_2^r \in \Phi_2(\pi_2^d)\}$$

Lemma 11 (Conformance of composition)**.** *Let R_1 and R_2 be two FSMr machines corresponding to features f_1, f_2, and let D_1 and D_2 be the corresponding FSMd machines. Let $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$. Let $\Phi = \Phi_1 + \Phi_2$ and π^d be a valid configuration of $D_1 \parallel D_2$. Then, $\forall \pi^r \in \Phi(\pi^d)$, $L([(D_1 \parallel D_2) \downarrow \pi^d]) \subseteq L([(R_1 \parallel R_2) \downarrow \pi^r])$.*

Proof. In Appendix D. □ □

Considering the example, in the FSMr $Req_{dl} \parallel Req_{du}$ with $\rho_r : DL\_Enable = DU\_Enable \wedge Transmission_{dl} = Transmission_{du}$, Any configuration where $DL\_Enable = Enable$ but $DU\_Enable = Disable$ is invalid. However, $\Phi(\langle Auto, Speed\rangle)$ contains only configurations where $DL\_Enable = Enable$, $\Phi'(\langle Moff, Poff\rangle)$ contains only configurations where $DU\_Enable = Disable$ and $\langle Auto, Speed\rangle + \langle Moff, Poff\rangle$ is a valid configuration of $Des_{dl} \parallel Des_{du}$. So the design does not conform to the requirement. However, if we make the extra assumption that $\rho_d : Cp1 = Moff \wedge Cp2 = Poff \Leftrightarrow Cp3 = Moff \wedge Cp4 = Poff$, then $\langle Auto, Speed\rangle$ and $\langle Moff, Poff\rangle$ are not compatible anymore and as a result the design conforms to the requirement.

3.1 Conformance Checking

Let $F = \{f_1, ..., f_n\}$ be a set of features and $\mathcal{F}$ be the complete system comprising the features in F, along with the relations between the features. Let R_i be the FSMr modeling the expected behavior and variability of f_i, and D_i the FSMd extracted from the design of f_i. Let $\rho_{12\ldots n}^r$ and $\rho_{12\ldots n}^d$ be the compositional predicates for $R_1 \parallel \cdots \parallel R_n$ and $D_1 \parallel \cdots \parallel D_n$ respectively. Now we state the variability conformance problem for an SPL as follows: Does there exist a conformance mapping Φ such that $D_1 \parallel \cdots \parallel D_n \leq_\Phi R_1 \parallel \cdots \parallel \ldots R_n$? A compositional approach to solve the problem is to:
(i) check whether the design of every feature conforms to its requirement using Algorithm 1; (ii) check whether every valid configuration of $D_1 \parallel \cdots \parallel D_n$ can be mapped to a valid configuration of $R_1 \parallel \cdots \parallel R_n$. This is the conformance condition.

3.2 Checking Conformance Using QBF

We implement the second check using QBF solving. Given FSMd's $D_1, \ldots, D_n$ and FSMr's $R_1, \ldots, R_n$,
(1) Let $Var(D_i) = \{v^d_{i1}, \ldots, v^d_{in}\}$ be the set of variables of design D_i, and $Var(R_i) = \{v^r_{i1}, \ldots, v^r_{im}\}$, the set of variables of requirement R_i. Let $\pi^d : (v^d_{i1} = a_1, \ldots, v^d_{in} = a_n)$ be a configuration of D_i. We denote by $\pi^d_i(x_{i1}, \ldots, x_{in})$ a formula which takes n values from $Dom(D_i), 1 \leq i \leq n$ as arguments. If $(v^d_{i1} = a_1, \ldots, v^d_{in} = a_n)$ is a chosen assignment, then $\pi^d_i(x_{i1}, \ldots, x_{in})$ is the conjunction $\bigwedge_{j=1}^{n}(x_{ij} = a_j)$;
(2) Given n FSMd's and n FSMr's check if D_i conforms to R_i for all $1 \leq i \leq n$ using Algorithm 1. This gives the map Φ_i. Assume $\Phi_i(\pi^d_i) = \{\pi^r_{i1}, \ldots, \pi^r_{im}\}$, where each of $\pi^r_{i1}, \ldots, \pi^r_{im}$ are configurations of R_i, that have been mapped by Φ_i to some configuration π^d_i of D_i.
(3) We encode the above conformance mapping using the formula $\Phi_i(x_{i1}, x_{i2}, \ldots, x_{in}) = \bigvee_{j=1}^{m} \pi^r_{ij}(y_{i1}, \ldots, y_{il})$, where x_{ij} takes values from $Dom(v^d_{ij})$, and y_{ij} from $Dom(v^r_{ij})$.
(4) Let $\varphi^d_{i,j} = \rho^d \wedge \rho^d_i \wedge \rho^d_j$ and $\varphi^r_{i,j} = \rho^r \wedge \rho^r_i \wedge \rho^r_j$ represent respectively the propositional formulae which ensures consistency of the global predicates of D_i, D_j and R_i, R_j along with the compositional predicates ρ^d and ρ^r. Given a set $S \subseteq \{1, 2, \ldots, n\}$, φ^d_S and φ^r_S can be appropriately written.
The QBF formula for conformance checking is given by

$$\Psi = \forall x_{11} \ldots x_{ni_n} [\varphi^d_{1,2,\ldots,n} \Rightarrow \exists y_{11} \ldots y_{nj_n} (\Phi_1 \wedge \cdots \wedge \Phi_n \wedge \varphi^r_{1,2,\ldots,n})]$$

Theorem 12. *Given a SPL, let $\{f_1, \ldots, f_n\}$ be the set of features in a chosen product. Let D_i, R_i be the FSMd and FSMr for feature f_i. Then $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$ iff Ψ holds.*

Proof. In Appendix E. □ □

4 Implementation and Case Studies

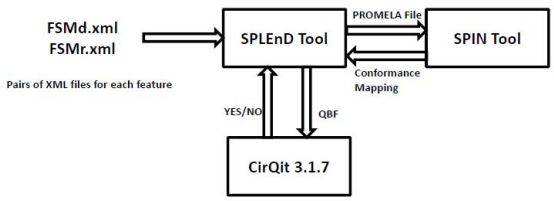


Figure 5: Overview of SPLEnD

Figure 5 pictorially describes the tool SPLEnD. It takes as input, a pair of xml files corresponding to FSMd, FSMr and outputs a PROMELA file. The latter is fed to SPIN, which returns the conformance mappings, or declares non-conformance; given the conformance mapping the tool computes a QBF formula Ψ which is fed to CirQit.

We considered two real case studies for our experimentation: Entry Control Product Line, ECPL having 7 features and Banking Software Product Line, BSPL, composed of 25 features. Appendices F.1 and F.2, contain the details of ECPL and BSPL case studies. The FSMr, FSMd models of each feature contains less than 10 states.

The analysis results for the two case studies are summarized in Figures 7 and 6 which gives the times taken by Algorithm 1. The number of product variants and the time taken for Algorithm 1 are very small in both case studies. In the case of ECPL, a bug was found in the feature *Door Lock* [2]. In this case, after fixing the bug, for the second step we used SPIN which took 11 seconds. For BSPL, the second step was performed using the QBF approach and CirQit took just 0.005 seconds.

In the automotive domain, really very large SPLs are constructed [8]. Before undertaking the task of modeling such large examples, in order to quickly determine the scalability of our approach, we generated many random SPLs with 5000 to 25,000 features. Each of the corresponding FSMr/FSMd has two variables (four variants), and 3 to 8 states. Similar to the ECPL and BSPL cases, SPIN took very little time (less than 0.5 seconds) for each (FSMr, FSMd) pair. The composite FSMr/FSMd, and hence the QBF formula Ψ has then 10,000 to 50,000 variables. As we can see from Figure 8, the the time taken for the largest example is 196.69 seconds which is quite efficient. Encouraged by this result, we plan to take up the large industrial case studies.

5 Conclusion

This paper motivated the need for extending the classical design verification problem to evolving SPL in which features and variability information can be added incrementally. The novel aspects of the proposed work are: (i) it verifies that the variability at the design level conforms to that at the requirement level, (ii) it is compositional and (iii) it reduces the conformance checking problem to QBF sat solving. A prototype tool has been implemented and experimented with modest sized examples with encouraging results.

References

[1] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information*

[2] In Des_{dl}, the transition from the middle elliptical state to the round state labeled with $Poff : ShiftOutOfPark$ is incorrect; $\Phi(\langle Auto, Poff\rangle) = \emptyset$. Removing this transition fixes the bug.

Sr. No.	Features	Design Variants	SPIN Time(Sec)
1	UserInterface	6	0.002
2	CheckingBalance	3	0.003
3	WithdrawMoney	8	0.027
4	DepositMoney	2	0.002
5	PrintingStatement	3	0.002
6	Login	1	0.001
7	ATMLogin	1	0.001
8	ChangeAccountPassword	2	0.003
9	PayBills	2	0.003
10	PrintingBalanceAfterWithdraw	2	0.003
11	CheckingMoneyExchangeRate	2	0.003
12	MoneyExchange	2	0.004
13	InternationalTransfer	2	0.006
14	LocalTransferToOtherBank	1	0.004
15	LanguageSelection	2	0.001
16	MobileTopUp	2	0.002
17	ChangeMaxLimitForWithdrawal	1	0.003
18	LocalTransferToSameBank	3	0.003
19	AddBeneficiary	1	0.002
20	RemoveBeneficiary	1	0.002
21	CreateDemandDraft	2	0.003
22	ChequeClearance	1	0.003
23	FastWithdrawal	1	0.002
24	CreditCardPayment	2	0.002
25	UpdateContactDetails	2	0.004

Figure 6: Execution time of FSMv-Verifier on Algorithm 1 for BSPL

Features	*PL & LDCL*	*PCU*	*DL*	*DU*	*AL*	*TSL*
Design Variants	8	3	4	7	3	8
SPIN Time (Sec)	0.436	0.031	0.046	0.109	0.015	0.218

Figure 7: Execution time of FSMv-Verifier on Algorithm 1 for ECPL

Variables in FSMr/FSMd	10000	20000	30000	40000	50000
CirQit 3.1.7 time (Sec)	4.47	25.77	65.67	119.49	196.69

Figure 8: Execution time of QBF for Scalability

Systems, 35(6):–, 2010.

[2] Kathrin Berg, Judith Bishop, and Dirk Muthig. Tracing software product line variability: from problem to solution space. In *SAICSIT '05*, pages 182–191. South African Institute for Computer Scientists and Information Technologists, 2005.

[3] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, Hawaii, Proceedings*, pages 321–330. ACM, 2011.

[4] Maxime Cordy, Andreas Classen, Gilles Perrouin, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Simulation relation for software product lines: Foundations for scalable model checking (to appear). In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland, Proceedings*, June 2012.

[5] Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, and Axel Legay. Behavioural modelling and verification of real-time software product lines. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 66–75, New York, NY, USA, 2012. ACM.

[6] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[7] Alessandro Fantechi and Stefania Gnesi. Formal modeling for product families engineering. In *SPLC*, pages 193–202, 2008.

[8] Rick Flores, Charles Krueger, and Paul Clements. Mega-scale product line engineering at general motors. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 259–268, New York, NY, USA, 2012. ACM.

[9] Hassan Gomaa and Erika Olimpiew. Managing variability in reusable requirement models for software product lines. In Hong Mei, editor, *High Confidence Software Reuse in Large Systems*, volume 5030 of *Lecture Notes in Computer Science*, pages 182–185. Springer Berlin / Heidelberg, 2008.

[10] Alexandra Goultiaeva and Fahiem Bacchus. Exploiting qbf duality on a circuit representation. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[11] Alexander Gruler, Martin Leucker, and Kathrin D. Scheidemann. Calculating and modeling common parts of software product lines. In *SPLC*, pages 203–212, 2008.

[12] Alexander Gruler, Martin Leucker, and Kathrin D. Scheidemann. Modeling and model checking software product lines. In Gilles Barthe and Frank S. de Boer, editors, *FMOODS*, volume 5051 of *Lecture Notes in Computer Science*, pages 113–131. Springer, 2008.

[13] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[14] Shriram Krishnamurthi and Kathi Fisler. Foundations of incremental aspect model-checking. *ACM Trans. Softw. Eng. Methodol.*, 16(2):39, 2007.

[15] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal i/o automata for interface and product line theories. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2007.

[16] Kim Lauenroth, Andreas Metzger, and Klaus Pohl. Quality assurance in the presence of variability. Technical report, SSE, Institut fur Informatik und Wirtschaftsinformatik, univertitat Duisburg Essen, 2011.

[17] Jing Liu, Samik Basu, and Robyn Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18:39–76, 2011. 10.1007/s10515-010-0075-7.

[18] Andreas Metzger and Klaus Pohl. Variability management in software product line engineering. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 186–187, Washington, DC, USA, 2007. IEEE Computer Society.

[19] Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 243–253, 2007.

[20] Jean-Baptiste Raclet, Benoît Caillaud, Eric Badouel, Axel Legay, Albert Benveniste, and Roberto Passerone. Modal interfaces: Unifying interface automata and modal specifications. In Christoph M. Kirsch and Reinhard Wilhelm, editors, *EMSOFT*. ACM, 2009.

[21] Matthias Riebisch and Robert Brcina. Optimizing design for variability using traceability links. In *ECBS '08: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 235–244, Washington, DC, USA, 2008. IEEE Computer Society.

[22] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In Cambridge, editor, *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science*, pages 322–331, June 1986.

Appendix

A Conformance Checking using SPIN

Algorithm 1 starts by generating a Promela file containing the definition of (i) the environment, (ii) Des_f, (iii) Req_f, (iv) the initialization sequence and (v) a never claim which holds for the language containment condition. During the initialization, the configuration of Des_f and Req_f are initialized with a random couple of configurations. Then the environment, followed by Des_f and Req_f are run atomically. The never claim assertion is : $never((\neg error(Des_f) \wedge error(Req_f))$, where $error(X)$ means that X is in error state. The never claim is violated when the design is not in the error state but the requirement process is in the error state. This corresponds to a design configuration π_d such that $Des_f \downarrow \pi_d$ handles an event, while $Req_f \downarrow \pi_r$ does not, for all possible requirement configurations π_r. Algorithm 1 runs the full verification algorithm of SPIN for every pair (π_d, π_r) of design and requirement configurations. SPIN(i.e. *pan(.exe)*) returns the list of pairs for which the conformance condition is violated. Every other pair is added to the conformance mapping Φ.

A.1 Proof of Lemma 5

Assume $L(Des_f \downarrow \pi_d) \not\subseteq L(Req_f \downarrow \pi_r)$. Then there exists a word $w \in L(Des_f \downarrow \pi_d)$ which is prefixed by $u.e$, with u a finite prefix of a word in $L(Req_f \downarrow \pi_r)$, and e an event such that $u.e$ is not a prefix of any word in $L(Req_f \downarrow \pi_r)$. In such a situation, Des_f does not go to the error state but Req_f does.

Conversely, if $L(Des_f \downarrow \pi_d) \subseteq L(Req_f \downarrow \pi_r)$, then whenever Des_f is not in an error state, Req_f will also not be in an error state. □

B Proof of Lemma 8

Let $\pi \in \Pi(\rho)$ with $\rho = \rho_{12} \wedge \rho_1 \wedge \rho_2$ be a valid configuration of $\mathcal{A}_1 \parallel \mathcal{A}_2$. ρ_1 and ρ_2 are the global predicates of $\mathcal{A}_1$, $\mathcal{A}_2$ respectively, and ρ_{12} is the composition predicate of $\mathcal{A}_1$, $\mathcal{A}_2$. By definition of valid configuration, $\pi \models \rho$; hence $\pi \models \rho_1$ and $\pi \models \rho_2$. Since π is a configuration over $Var_1 \cup Var_2$, let us consider the restriction of π on Var_1, call the resulting configuration π_1. Then $\pi_1 \models \rho_1$. Similarly, call the restriction of π on Var_2 as π_2. Then $\pi_2 \models \rho_2$. Then, π_1, π_2 are respectively valid configurations of $\mathcal{A}_1$ and $\mathcal{A}_2$. Hence, by definition 7, we obtain $\pi = \pi_1 + \pi_2$. □

C Proof of Lemma 9

We review some preliminary definitions before the proof. In the following, the operation $\parallel$ stands for (i) shuffle of words, (ii) shuffle of languages, (iii)parallel

composition of FSMs, and (iv) parallel composition of FSMv. The context is clear in each case; hence there is no confusion.

Definition 13. *Let $\Sigma_1, \ldots, \Sigma_n$ be n finite sets of symbols. Let Σ be a finite set. Given a word $w \in \Sigma^*$, we denote by $w \downarrow \Sigma_i$, the unique subword of w over Σ_i^*. For example, if $\Sigma_1 = \{a, b, e\}, \Sigma_2 = \{a, e, f\}$, and if we consider $w = aefedefr \in \{a, d, e, f, r\}^*$, then $w \downarrow \Sigma_1 = aeee$ and $w \downarrow \Sigma_2 = aefeef$.*

Definition 14. *(Asynchronous Shuffle) Let $\Sigma_1, \ldots, \Sigma_n$ be n finite sets. Let $\Sigma = \cup_{i=1}^{n} \Sigma_i$. Consider n words $u_1, u_2, \ldots, u_n$, $u_i \in \Sigma_i^*$. The asynchronous shuffle of $u_1, \ldots, u_n$ denoted $u_1 \parallel \cdots \parallel u_n$ is defined as $\{w \mid w \downarrow \Sigma_i = u_i\}$.*

As an example, consider $\Sigma_1 = \{a, b, c, f\}, \Sigma_2 = \{a, d, e, f\}, \Sigma_3 = \{c, d, f\}$, and the words $u_1 = abcf, u_2 = adfe, u_3 = dcf$. Then the word $w = abdcfe$ is in $u_1 \parallel u_2 \parallel u_3$ since, $w \downarrow \Sigma_i = u_i$ for $i = 1, 2, 3$. Similarly, the word $w' = adbcfe$ is also in $u_1 \parallel u_2 \parallel u_3$. However, the word $w'' = aebcfd$ is not in $u_1 \parallel u_2 \parallel u_3$, since $w'' \downarrow \Sigma_2 = aefd$, not u_2.

The definition of shuffle can be extended from words to languages. We use the same notation $\parallel$ for the shuffle of sets, as well as for the shuffle of words.

The asynchronous shuffle of two languages L_1, L_2 is defined as $L_1 \parallel L_2 = \{w_1 \parallel w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. For example, if $L_1 = \{abcf, abbf\}$ is a language over $\Sigma_1 = \{a, b, c, f\}$ and $L_2 = \{adfe\}$ is a language over $\{a, d, e, f\}$, then $L_1 \parallel L_2 = \{abcf \parallel adfe, abbf \parallel adfe\} = \{abcdfe, adbcfe, abdcfe, abbdfe, abdbfe, adbbfe\}$.

Definition 15. *Let $M_i = (Q_i, q_i, \Sigma_i, \delta_i)$ and $M_j = (Q_j, q_j, \Sigma_j, \delta_j)$ be complete FSMs. The asynchronous product of M_i, M_j is defined as the FSM $M_i \parallel M_j = (Q_i \times Q_j, (q_i, q_j), \Sigma_i \cup \Sigma_j, \delta)$ where*

1. $\delta((q, q'), a) = (\delta_i(q, a), \delta_j(q', a)), a \in \Sigma_i \cap \Sigma_j$,
2. $\delta((q, q'), a) = (\delta_i(q, a), q'), a \in \Sigma_i, a \notin \Sigma_j$,
3. $\delta((q, q'), a) = (q, \delta_j(q', a)), a \in \Sigma_j, a \notin \Sigma_i$.

On the common events, both FSMs move in parallel; otherwise, they move independent of each other.

It is known that $L(M_i \parallel M_j) = L(M_i) \parallel L(M_j)$. Now we start the proof of Lemma 9.
Consider a valid configuration π of $\mathcal{A}_1 \parallel \mathcal{A}_2$. As seen in Lemma 8, we can find valid configurations π_1 of $\mathcal{A}_1$ and π_2 of $\mathcal{A}_2$ such that $\pi = \pi_1 + \pi_2$. The initial state of $\mathcal{A}_1 \parallel \mathcal{A}_2$ is (q_0^1, q_0^2), where q_1^0 is the initial state of $\mathcal{A}_1$ and q_2^0 is the initial state of $\mathcal{A}_2$. By definitions 6 and 15, if we consider a string $w = a_1 a_2 \ldots a_n \in L[\mathcal{A}_1 \parallel \mathcal{A}_2] \downarrow \pi$, then we can find strings $w_1 \in L(\mathcal{A}_1 \downarrow \pi) = L(\mathcal{A}_1 \downarrow \pi_1)$ and $w_2 \in L(\mathcal{A}_2 \downarrow \pi) = L(\mathcal{A}_2 \downarrow \pi_2)$ such that $w = w_1 \parallel w_2$ in the sense of definition 14. Hence, $L[\mathcal{A}_1 \parallel \mathcal{A}_2] \downarrow \pi \subseteq L(\mathcal{A}_1 \downarrow \pi) \parallel L(\mathcal{A}_2 \downarrow \pi)$. The converse can be shown in a similar way. □

D Proof of Lemma 11

Given a valid configuration π^d of $D_1 \parallel D_2$, we can write it as $\pi_1^d + \pi_2^d$, where π_1^d, π_2^d are respectively valid configurations of D_1, D_2 (Lemma 8). Since $D_1 \leq_{\Phi_1} R_1$ and $D_2 \leq_{\Phi_2} R_2$, there exist valid configurations $\pi_1^r \in \Phi_1(\pi_1^d)$ and $\pi_2^r \in \Phi_2(\pi_2^d)$ such that $L(D_1 \downarrow \pi_1^d) \subseteq L(R_1 \downarrow \pi_1^r)$ and $L(D_2 \downarrow \pi_2^d) \subseteq L(R_2 \downarrow \pi_2^r)$.

Since Φ has been computed, for every valid configuration π^d of $D_1 \parallel D_2$, there exists some valid configuration π^r of $R_1 \parallel R_2$, $\pi^r \in \Phi(\pi^d)$. As π^r is valid, $\pi^r \models \rho_{12}^r \wedge \rho_1^r \wedge \rho_2^r$; hence, π^r can be written as $\pi_1^r + \pi_2^r$, where π_1^r, π_2^r are respectively valid configurations of R_1, R_2 (Lemma 8), and $\pi_1^r \in \Phi_1(\pi_1^d)$, $\pi_2^r \in \Phi_2(\pi_2^d)$ by definition 10.

$L([(D_1 \parallel D_2) \downarrow \pi^d]) = L(D_1 \downarrow \pi_1^d) \parallel L(D_2 \downarrow \pi_2^d)$ by lemma 9. Similarly, $L([(R_1 \parallel R_2) \downarrow \pi^r]) = L(R_1 \downarrow \pi_1^r) \parallel L(R_2 \downarrow \pi_2^r)$. This along with the observation that $L(D_1 \downarrow \pi_1^d) \subseteq L(R_1 \downarrow \pi_1^r)$ and $L(D_2 \downarrow \pi_2^d) \subseteq L(R_2 \downarrow \pi_2^r)$ gives $L([(D_1 \parallel D_2) \downarrow \pi^d]) \subseteq L([(R_1 \parallel R_2) \downarrow \pi^r])$. □

E Proof of Theorem 12

Given $D_i \leq_\Phi R_i$, assume that $D_1 \parallel \cdots \parallel D_n$ conforms to $R_1 \parallel \cdots \parallel R_n$. Then, by definition of conformance, it means that for all valid configurations π^d of $D_1 \parallel \cdots \parallel D_n$, there exists a valid configuration π^r of $R_1 \parallel \cdots \parallel R_n$ such that $L([D_1 \parallel \cdots \parallel D_n] \downarrow \pi^d) \subseteq L([R_1 \parallel \cdots \parallel R_n] \downarrow \pi^r)$. Let Φ be the conformance mapping such that $\pi^r \in \Phi(\pi^d)$.

π^d is a valid configuration of $D_1 \parallel \cdots \parallel D_n$ implies that $\pi^d \models \bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho_S^d$, where ρ_S^d is the global predicate of $D_{i_1} \parallel \cdots \parallel D_{i_j}$, when $S = \{i_1, \ldots, i_j\}$. Using Lemma 8 repeatedly, we can then say that $\pi^d = \pi_1^d + \cdots + \pi_n^d$ for valid configurations π_i^d of D_i. Since $\pi^r \in \Phi(\pi^d)$, by definition of conformance mappings, π^r must be a valid configuration of $R_1 \parallel \cdots \parallel R_n$, hence $\pi^r = \pi_1^r + \cdots + \pi_n^r$ (Lemma 8), such that $\pi_i^d \in \Phi(\pi_i^r)$, for valid configurations π_i^r of R_i. π^r is valid means $\pi^r \models \bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho_S^r$.

Given the above, we show that the QBF Ψ holds. The LHS of the QBF Ψ is the formula $\varphi_{1,2,\ldots,n}^d$, which is the conjunction ρ_S^d for all subsets S of $\{1, 2, \ldots, n\}$. The forall quantifier outside would thus evaluate all configurations of $D_1 \parallel \cdots \parallel D_n$ that satisfy $\varphi_{1,2,\ldots,n}^d$; that is, which satisfy $\bigwedge_{S \subseteq \{1,2,\ldots,n\}} \rho_S^d$: hence, all valid configurations of $D_1 \parallel \cdots \parallel D_n$.

For the QBF to hold good, for all valid configurations of $D_1 \parallel \cdots \parallel D_n$ that have been evaluated on the LHS, we must find some configuration of $R_1 \parallel \cdots \parallel R_n$ that satisfies $\Phi_1 \wedge \cdots \wedge \Phi_n \wedge \varphi_{1,2,\ldots,n}^r$: (i) any configuration π of $R_1 \parallel \cdots \parallel R_n$ that satisfies $\varphi_{1,2,\ldots,n}^r$ would be valid; (ii) further, if it has to satisfy $\Phi_1 \wedge \cdots \wedge \Phi_n$, it must agree with $\pi_i^r \in \Phi_i(\pi_i^d)$ over $Var(R_i)$ for all $1 \leq i \leq n$. By Lemma 8, this means that π can be written as $\pi_1^r + \cdots + \pi_n^r$. Thus, for the QBF to hold, we must be able to find for each valid configuration π^d of $D_1 \parallel \cdots \parallel D_n$, a valid configuration π^r of $R_1 \parallel \cdots \parallel R_n$ which can be written as $\pi_1^r + \cdots + \pi_n^r$, where $\pi_i^r \in \Phi_i(\pi_i^d)$ for each i. But this is exactly what the mapping Φ which checks

for conformance of $D_1 \parallel \cdots \parallel D_n$ with $R_1 \parallel \cdots \parallel R_n$ does. Since we assume that Φ exists, the QBF holds.

The converse can be shown in a similar way : that is, if the QBF formula Ψ holds, then $D_1 \parallel \cdots \parallel D_n$ will conform to $R_1 \parallel \cdots \parallel R_n$. □

F Case Studies

In this section, we describe the two product lines that have been considered in the paper : (i) ECPL and (ii) BSPL.

F.1 ECPL

The Entry Control Product Line comprises all the features involved in the management of the locks in a car. In this study, we focus on the following features:

- *Power lock*: this is the basic locking functionality which manages the locking/unlocking according to key button press and courtesy switch press,
- *Last Door Closed Lock*: delays the locking of the doors until all the doors are closed. It is applicable when the lock command appends while a door is open,
- *Door lock*: automates the locking of doors when the vehicle starts,
- *Door unlock*: automates the unlocking of door(s) when the vehicle stops,
- *Anti-lockout*: is intended to prevent the inadvertent lockout situations: the driver is out of the car with the key inside and all the doors locked,
- *Post crash unlock*: unlocks all the doors in a post crash situation,
- *Theft security lock*: secures the car with a second lock.

Each feature is represented as a pair of state machines containing 3 to 10 states.

F.1.1 The variability constraints of the ECPL

Figure 9 presents the feature diagram of the ECPL (a la Czarnecki [6]). This diagram presents the variability constraints of the ECPL at the requirement level (ρ_{f_0}). All the constraints represented by this diagram have to be considered during composition to guarantee the overall consistency of the SPL behavior. The dark gray boxes are features of the ECPL: *Power lock*, *Anti-lockout*, *Door lock*, *Door unlock*, and *Post crash unlock*. The light gray boxes are configurations. The black arrow from the “Manual" configuration to the “Shift out of park" configuration and to the “Shift into park" configuration says that if the transmission is manual, the targeted configurations cannot be selected. i.e. In “Manual" configuration, there is no “park" gear.

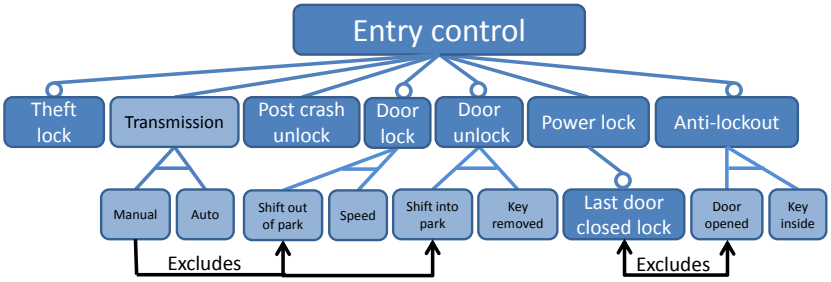


Figure 9: The feature diagram of the ECPL.

F.2 BSPL

The Banking Software Product Line (BSPL) consists of 25 behavioral features. The BSPL is used to derived the software for ATM, Bank, Online Banking and Mobile Banking. Figure 10 presents the feature diagram of the BSPL.

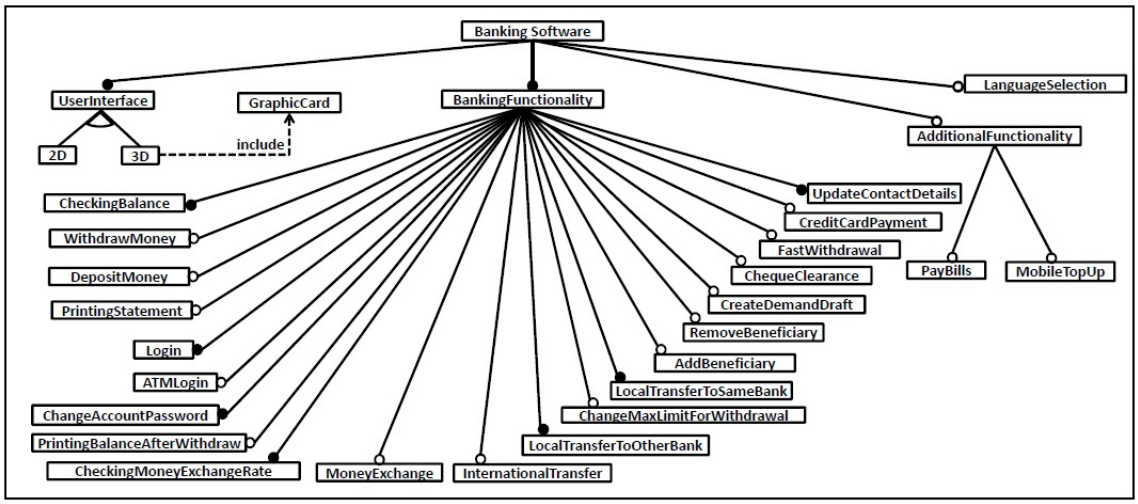


Figure 10: The feature diagram of the BSPL.

Similar to ECPL, we ran Algorithm 1 on all the 25 features of BSPL. In section 4, Figure 6 presents the number of design configurations and execution time of Algorithm 1 for each feature. In the following, we elaborate on the FSMv of 2 features: (i) User Interface and (ii) Withdraw Money. The FSMd/FSMr

for all the features has states between 2 and 10 (both inclusive). Figure 11 is the FSMr for feature $User\ Interface$, which has UI as an event with global predicate $\rho = \{\neg(uip = Disable)\}$. There is only one boolean variable, $Var = \{uip\}$, uip takes values from $\{Enable, Disable\}$.

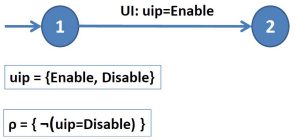


Figure 11: FSMr for feature: $UserInterface$.

Figure 12 is the FSMd for feature $User\ Interface$. This FSMd shares the event UI with the FSMr and has global predicate $\rho = \{(type = 2D \vee type = 3D)\}$. There are two variables, $Var = \{type, graphics\}$, $type$ takes values from $\{2D, 3D\}$, while $graphics$ takes values from $\{Enable, Disable\}$. The xml file

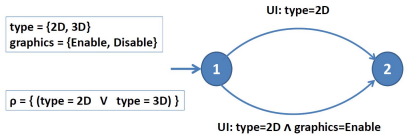


Figure 12: FSMd for feature: $UserInterface$.

corresponding to the FSMr in Figure 11 is as below:

```
<?xml version='1.0'>
<FSMv>
<type>R</type>
<name>UserInterface_req</name>
<states>
<s initial=true>1</s>
<s>2</s>
</states>
<set_of_sets_of_final_states>
```

```
<group_of_final_states>
<s>2</s>
</group_of_final_state>
</set_of_sets_of_final_states>
<events>
<e>UI</e>
</events>
<variables>
<variable>
<v_name>uip</v_name>
<value>Enable</value>
<value>Disable</value>
</variable>
</variables>
<rho>
<conjunct>not(uip=Disable)</conjunct>
</<rho>
<vds>
<predicate>
<id>1</id>
<equ>(uip=Enable)</equ>
</predicate>
</vds>
<transitions>
<t>
<start>1</start>
<end>2</end>
<vdid>1</vdid>
<events>UI</events>
</t>
</transitions>
</FSMv>
```

The xml file for the FSMd in Figure 12 is as below:

```
<?xml version='1.0'>
<FSMv>
<type>D</type>
<name>UserInterface_des</name>
<states>
<s initial=true>1</s>
<s>2</s>
</states>
<set_of_sets_of_final_states>
<group_of_final_states>
<s>2</s>
</group_of_final_state>
```

```
</set_of_sets_of_final_states>
<events>
<e>UI</e>
</events>
<variables>
<variable>
<v_name>type</v_name>
<value>2D</value>
<value>3D</value>
</variable>
<variable>
<v_name>graphics</v_name>
<value>Enable</value>
<value>Disable</value>
</variable>
</variables>
<rho>
<conjunct>or(type=2D,type=3D)</conjunct>
</<rho>
<vds>
<predicate>
<id>1</id>
<equ>(type=2D)</equ>
</predicate>
<predicate>
<id>2</id>
<equ>and(type=3D,graphics=Enable)</equ>
</predicate>
</vds>
<transitions>
<t>
<start>1</start>
<end>2</end>
<vdid>1</vdid>
<events>UI</events>
</t>
<t>
<start>1</start>
<end>2</end>
<vdid>2</vdid>
<events>UI</events>
</t>
</transitions>
</FSMv>
```

These two xml files are given as input to SPLEnD, which creates the PROMELA

model for the feature $User\ Interface$. This model is given as input to SPIN, which returns the conformance mapping Φ for the feature $User\ Interface$. Similarly, Φ is constructed for all other features. The PROMELA model for the FSMd in Figure 12 is as follows:

```
#define  d_type_2D 0
#define  d_type_3D 1

#define  d_graphics_Enable 0
#define  d_graphics_Disable 1



#define  r_uip_Enable 0
#define  r_uip_Disable 1



 /*The Events*/
#define  evt_UI 0


 /*The states of the design model*/
#define  des_1 0
#define  des_2 1
#define des_error 2


 /*The states of the requirement model*/
#define  req_1 0
#define  req_2 1
#define req_error 2



/*this channel is use to forward the event
in both des as well in req. from environment*/
chan evts_req= [0] of {byte};
chan evts_des= [0] of {byte};

/*State variable*/
byte req_state;
 byte des_state;

/*Initialization  variables*/
 byte vp_uip;
```

```
 byte vp_type;
 byte vp_graphics;
byte flag;



proctype environmentModel(){
  do
::flag==0 -> flag=1; atomic{if
::(1)-> evts_des! evt_UI;   evts_req!evt_UI;
fi;}
  od;
 };


proctype requirementModel() {
mtype currentEvent;
  req_state=req_1;
 do
 ::flag==2-> evts_req?currentEvent;
  if
::req_state== req_1-> if
::vp_uip==r_uip_Enable &&  currentEvent== evt_UI-> req_state=req_2;
::else -> req_state= req_error;
  fi;
::req_state== req_2-> if
::else -> req_state= req_error;
  fi;
::else -> req_state = req_error;
 fi;flag=0;
 od;
 };


proctype designModel() {
mtype currentEvent;
  des_state=des_1;
 do
 ::flag==1-> evts_des?currentEvent;
  if
::des_state== des_1-> if
::vp_type==d_type_2D &&  currentEvent== evt_UI-> des_state=des_2;
::(vp_type==d_type_3D && vp_graphics==d_graphics_Enable)
&&  currentEvent== evt_UI-> des_state=des_2;
::else -> des_state= des_error;
fi;
```

```
::des_state== des_2-> if
::else -> des_state= des_error;
fi;
 ::else -> des_state = des_error;
 fi;flag=2;
od;
 };
```

```
 /*never claim definintion*/
never {
 T0_init:
if
  ::(flag==0 && req_state==req_error && des_state!=des_error)
  ->printf("vp_uip: %d, vp_type: %d, vp_graphics: %d\n",
vp_uip, vp_type, vp_graphics);
 goto accept_S9
::(1) -> goto T0_init
fi;
accept_S9:
if
::(1) -> goto T0_init
fi;
 }
```

```
init{
 flag=0; atomic{ if
:: (1)-> vp_uip=r_uip_Enable;
fi;}
atomic{
 if
:: (1)->vp_graphics=d_graphics_Enable ; vp_type=d_type_2D ;
  :: (1)->vp_graphics=d_graphics_Disable ; vp_type=d_type_2D ;
  :: (1)->vp_graphics=d_graphics_Enable ; vp_type=d_type_3D ;
  :: (1)->vp_graphics=d_graphics_Disable ; vp_type=d_type_3D ;
  fi;}
atomic {
  run environmentModel();
run requirementModel();
run designModel();
 }
```

}

Figure 13 is the FSMr for the feature $Withdraw\ Money$, which has events $\{WD, enterAmount, insuffUserBal, infussATMBal, checkATMBal, Disburse, dispatch\}$ and global predicate $\rho = \{(WDO = Enable \wedge \neg(med = Online)\}$. There are two variables, $Var = \{WDO, med\}$, WDO can take values from $\{Enable, Disable\}$, while med takes values from $\{Bank, ATM, Online\}\}$.

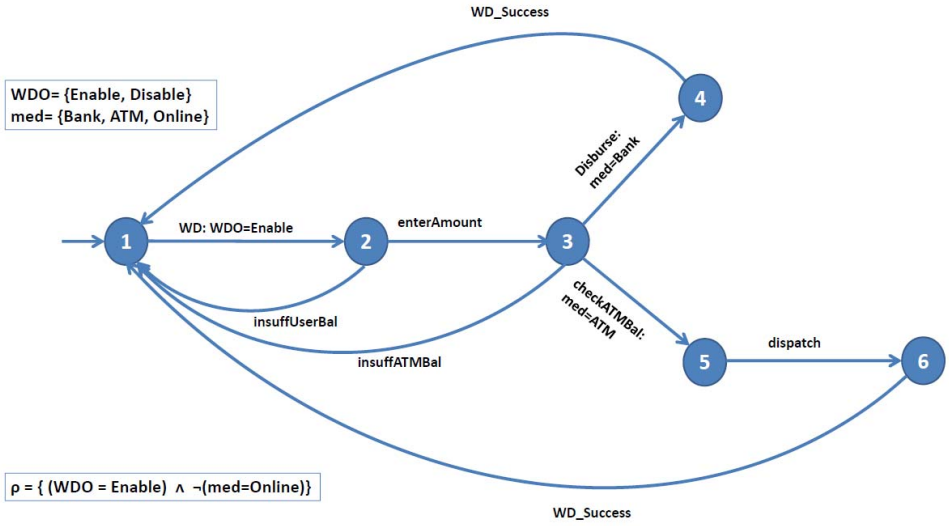


Figure 13: FSMr for feature: $WithdrawMoney$.

Figure 14 is the FSMd for the feature $Withdraw\ Money$. This FSMd shares all the events $\{WD$, $enterAmount$, $insuffUserBal$, $infussATMBal$, $checkATMBal$, $Disburse$, $dispatch\}$ of the FSMr. The global predicate is $\rho = \{(WDO = Enable \wedge \neg(med = Online)\}$. There are six variables, $Var = \{WDO, med, p1, p2, p3, p4\}$. WDO takes values from $\{Enable, Disable\}$, med takes values from $\{Bank, ATM, Online\}$, $p1$ to $p4$ are boolean variables which assume the value true depending on whether a predicate is satisfied or not. These predicates can be seen in the box on the top left of Figure 14.

After running SPLEnD on the feature $Withdraw\ Money$, we get the illegal configuration in design as $\pi = \{WDO = Enable, med = ATM, p1 = True, p2 =$

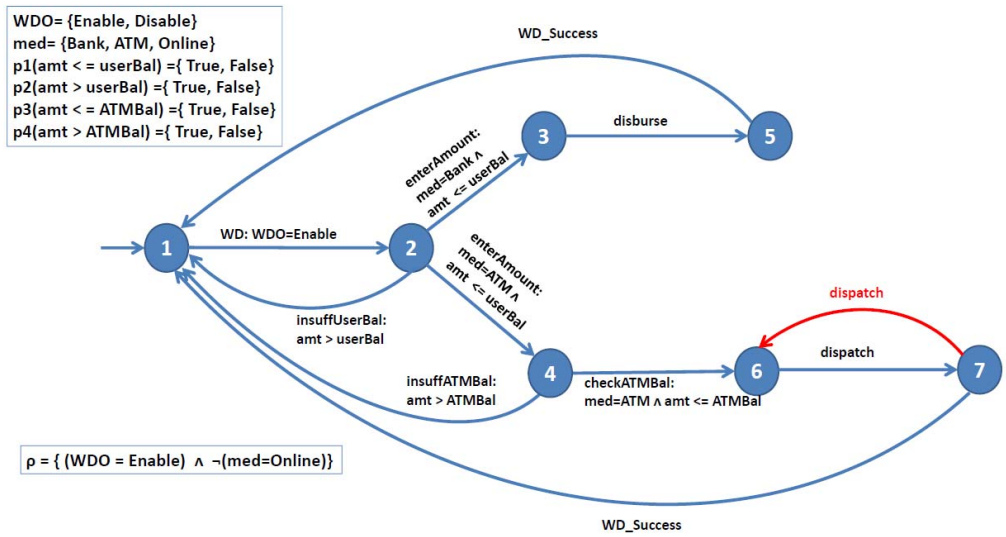


Figure 14: FSMd for feature: $WithdrawMoney$.

$False, p3 = True, p4 = False\}$. When we project this configuration on the FSMd, we get an FSM as shown in Figure 15.

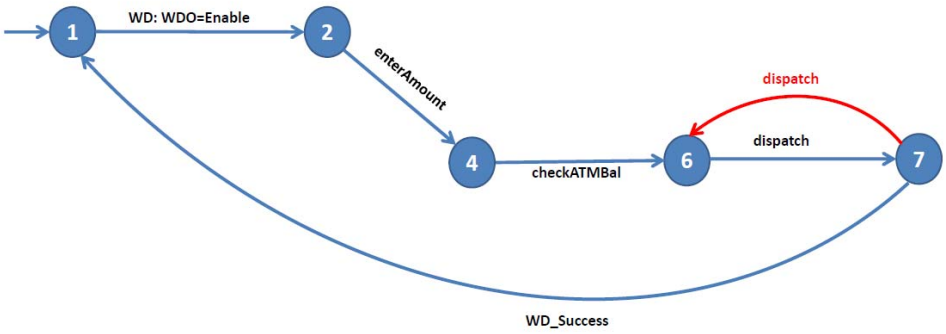
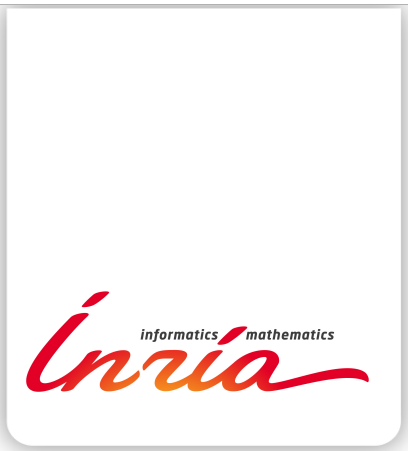


Figure 15: FSM for configuration π on FSMd.

It can be seen that the language of this FSM cannot be contained in any of the variants of the FSMr in Figure 13. So, we removed the transition in red from Figure 14, to obtain conformance. With this modified FSMd, we carried out the rest of the study. After construction of Φ for all features, SPLEnd created the QBF for the composite FSMd's and FSMr's. This QBF is converted to 'qpro' format which is an input format for CirQit QBF solver. CirQit verified this QBF formula, and returned the results to SPLEnD. The time taken by CirQit was 0.005 seconds.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399